Partial Order Techniques for Distributed Discrete Event Systems: why you can't avoid using them

Eric Fabre, Albert Benveniste

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1800

PARTIAL ORDER TECHNIQUES FOR DISTRIBUTED DISCRETE EVENT SYSTEMS: WHY YOU CAN'T AVOID USING THEM

ERIC FABRE , ALBERT BENVENISTE

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

Institut de Recherche en Informatique et Systèmes Aléatoires
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

Partial Order Techniques for Distributed Discrete Event Systems: why you can't avoid using them*

Eric Fabre , Albert Benveniste **

Systèmes communicants
Projet DistribCom

Publication interne n ° 1800 — May 2006 — 32 pages

Abstract: Monitoring or diagnosis of large scale distributed Discrete Event Systems with asynchronous communication is a demanding task. Ensuring that the methods developed for Discrete Event Systems properly scale up to such systems is a challenge. In this paper we explain why the use of partial orders cannot be avoided in order to achieve this objective.

To support this claim, we try to push classical techniques (parallel composition of automata and languages) to their limits. We focus on on-line techniques, where a key difficulty is the choice of proper data structures to represent the set of all runs of a distributed system. We discuss the use of previously known structures such as execution trees and unfoldings. We propose an alternative and more compact data structure called *trellis*.

We study the apparatus needed to extend the use of these data structures to represent distributed executions. And we show how such data structures can be used in performing distributed monitoring and diagnosis.

The techniques reported here were used in an industrial context for fault management and alarm correlation in telecommunications networks.

This report served as a support for a plenary address that was given by the second author at WODES'2006.

Key-words: Discrete Event Systems, distributed systems, diagnosis, partial orders, unfoldings, fault management

(Résumé : tsvp)

* This report has been written as a support to the plenary address given by the second author at WODES 2006. This work has been supported in part by joint RNRT contracts Magda and Magda2, with France Telecom R&D and Alcatel, funded by french Ministère de la Recherche, and by direct contracts with Alcatel. This paper reports on experience and joint work with Stefan Haar and Claude Jard, from IRISA. It is based on tight cooperation and interaction with Chistophe Dousson from France Telecom R&D and Armen Aghasaryan from Alcatel.

** IRISA-INRIA, Campus de Beaulieu, 35042 Rennes; surname.name@inria.fr

Centre National de la Recherche Scientifique (UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique et en Automatique – unité de recherche de Rennes

Techniques d'ordres partiels pour les systèmes à événements discrets répartis: pourquoi ne peut-on s'y soustraire

Résumé : Ce document explique pourquoi l'on ne peut échapper au recours aux modèles d'ordres partiels pour l'algorithmique des systèmes à événements discrets répartis. Dans le présent document nous nous en tenons à la surveillance et ne traitons pas du contrôle. Les techniques présentées ont été utilisées dans un contexte industriel, pour la gestion répartie d'alarmes dans les réseaux de télécommunications.

Mots clés : systèmes à événements discrets, systèmes répartis, diagnostic, ordres partiels, dépliages, gestion d'alarmes

Contents

1 Introduction

Since the pioneering work by Ramadge and Wonham, the Discrete Event Systems (DES) community has developed a rich body of frameworks, techniques, and algorithms for the supervision of DES. While most authors have considered supervision of a monolithic automaton or language, decentralized frameworks have been more recently considered [7]–[11] and [21, 32].

While different architectures have been studied by these authors, the type of situation considered is the following: The system considered is observed by a finite set of *agents*, indexed by some finite index set I. Agent i can observe events labeled by some subalphabet $L_i \subset L$ of the message alphabet. Local decisions performed by the local agents are then forwarded to some central supervisor, which takes the final decision regarding observation; decisions mechanisms available to the supervisor are simple policies to combine the decisions forwarded by the local agents, *e.g.,* conjunction, disjunction, etc [32]. Of course, there is no reason why such decentralized setting should be equivalent to the centralized one. Therefore, various notions of decentralized observability, controllability, and diagnosability have been proposed for each particular architecture, see *e.g.,* [32]. Deciding upon such properties can then become infeasible [31].

Whereas these are important results, they fail to address the issue of large systems, where global model, global state, and sometimes even global time, should be avoided. Accordingly, in this work, we consider a distributed system $\mathcal{A}$ with subsystems $\mathcal{A}_i, i \in I$ and a set of sensing systems $\mathcal{O}_i, i \in I$ attached to each subsystem. The goal is to perform the monitoring of $\mathcal{A}$ under the following constraints:

- a supervisor $\mathcal{D}_i$ is attached to each subsystem;
- supervisor $\mathcal{D}_i$ does not know the global system model $\mathcal{A}$; it only knows a local view of $\mathcal{A}$, consisting of $\mathcal{A}_i$ plus some interface information relating $\mathcal{A}_i$ to its neighbors;
- supervisor $\mathcal{D}_i$ accesses observations made by $\mathcal{O}_i$;
- the different supervisors act as peers; they can exchange messages with their neighboring supervisors; they concur at performing system monitoring;
- no global clock is available, and the communication infrastructure is asynchronous.

Fewer results are available on DES monitoring that comply with these requirements. In this paper, we shall first try to solve this problem in the classical framework of automata, languages, and their parallel composition; we refer to this as the *sequential framework* since time and states are global, and runs are sequences of events. To account for our distributed setting, we avoid manipulating global models (we stick with products of automata instead), and we avoid manipulating global runs (we stick with sets of synchronized local runs instead).

Such type of study has, for example, been performed by Su and Wonham [28, 29, 30] by developing a distributed algorithm involving supervising peers that exchange messages to achieve either *local consistency* (the peers agree on the restrictions of their monitoring

results to their common interfaces) or *global consistency* (each peer actually computes the local projection of the global monitoring result). Their algorithm manipulates languages, *i.e.,* sets of runs. Similar work has been performed independently by [25] and [26].

To properly scale up, the issue of efficiently representing these sets of runs must be addressed, see [6], [14]–[19], and [4]. This is the main focus of the present paper.

We first investigate the use of *execution trees* to represent sets of runs of an automaton. We show how to represent monitoring or diagnosis, both off-line and on-line, in terms of such execution trees. We then show how execution trees can be factorized with a proper notion of product, when the automaton itself is a product. And we show how this product of execution trees can be computed in a distributed way, by using a *belief propagation* type of algorithm involving chaotic and asynchronous exchanges of messages between the supervising peers. We also show how this can be performed on-line, while observations are collected by the peers. Instrumental in performing this are some key operators on execution trees, namely: *intersection, projection, and product.*

Still, this is not entirely satisfactory: even though execution trees are local, they grow exponentially with observation length. We thus propose to reuse an old idea from control, namely *trellises* of runs such as used in the classical dynamic programming or Viterbi algorithms. In a trellis, the set of all runs is represented by 1/ superimposing common prefixes, and 2/ merging futures of runs that reach identical states and have identical length for their past (*i.e.,* have explained the same number of events in their past). Intersections and products can be easily defined for trellises. Unfortunately, *while products and intersections can properly be defined for trellises, projections cannot.* As a consequence, no distributed algorithm can be developed with trellises as above.

The very problem is that, while defining trellises in the classical way, we use a *global* counting criterion for merging futures of runs. The solution is to replace global counters by *multi-counters*, *i.e.,* to have one counter for each peer. This is not a new idea in fact, as *multi-clocks,* also called *vector clocks,* have been introduced in the 80's by computer scientists [24, 20] to keep track of consistent global states by distributed peers, in the context of distributed systems. With multi-counters, all the needed apparatus works for trellises (intersection, projection, and product) and distributed on-line monitoring algorithms can be developed. These algorithms use much less memory than those using execution trees, not to speak about those manipulating languages directly. In fact, other valid criteria for merging futures can be used as well. Requirements for such criteria is that they project well onto the components; this obviously holds for multi-counters, but not for global counters.

> *The bottom line is that the right picture for a global run is rather a set of synchronized local runs, each local run possessing its own local criterion for merging futures. This means that a partial order view of global executions is indeed required.*

Of course, if internal concurrency also exists within each individual subsystem, then using partial order setting also within each local trellis is recommended. We will discuss the difficulties in doing this, see also [17] and [4].

The paper is organized as follows. Related work is discussed in Section 2; closest work to ours is that of Su and Wonham [28, 29, 30]. In Section 3 we investigate distributed diagnosis in the classical framework of automata equipped with parallel composition. Sets of runs as well as diagnoses are represented as execution trees, also called unfoldings. A distributed diagnosis algorithm is presented, where the supervisors act as peers by exchanging messages, asynchronously. This algorithm suffers from an excessive size of data structures while performing diagnosis: unfoldings are not compact enough. Eric Fabre proposed using trellis instead, a more compact data structure already considered in the control community in the context of dynamic programming and Viterbi algorithm. We investigate the use of this more efficient data structure in Section 4. In particular we explain why a proper use of it requires kind of a partial order view of distributed executions. How to move to a full fledged partial order viewpoint for diagnosis is discussed in Section 5. Related problems that must be considered but are not discussed in this paper are briefly listed in Section 6. Finally, Appendix A reports on our application experience in the context of fault management in telecommunications networks and services.

No proofs are given in this tutorial paper, proper references are given for these.

2 Discussing related work

Following the classical setting, we model our system for monitoring as an automaton $\mathcal{A} = (S, L, \rightarrow, s_0)$, where S is the set of states, L is the set of labels, and $s \xrightarrow{\ell} s'$ is the transition relation. Call *run* a sequence of successive transitions: $\sigma : s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} s_2 \ldots$ and denote by $\Sigma_{\mathcal{A}}$ the set of all runs of $\mathcal{A}$. Partition L as $L = L_o \cup L_u$, where L_o and L_u are the observed and unobserved labels, respectively, and let $\mathbf{Proj}_o(\sigma)$ be the *visible projection* of σ obtained by erasing unobserved labels from σ, and replacing states by a counting of observed labels (see Fig. 2). Denote by

$$\Sigma_{\mathcal{A},o} = \{\mathbf{Proj}_o(\sigma) \mid \sigma \in \Sigma_{\mathcal{A}}\}$$

the set of all *observations* of $\mathcal{A}$. The *monitor* of $\mathcal{A}$ is an algorithm that computes, for every observation $O \in \Sigma_{\mathcal{A},o}$, the set of runs explaining O, namely:

$$\mathbf{Proj}_o^{-1}(O) \tag{1}$$

2.1 The pre-computed approach

This approach addresses a weaker version of the monitor : only final states of runs in $\mathbf{Proj}_o^{-1}(O)$ are of interest, rather than the complete runs explaining O. The monitor takes the form of a "compiled" algorithm, meaning that a structure is statically computed (specifically a deterministic automaton) that, when fed with observation O, delivers the desired solution. This is simple and well known. With notations as above, the algorithm is:

1. Compute the *invisible reach* $\mathcal{A}_{/L_u}$ by hiding, in $\mathcal{A}$, labels belonging to L_u; $\mathcal{A}_{/L_u}$ is in general nondeterministic.

2. Determinize $\mathcal{A}_{/L_u}$.

This is known in the DES literature as an *observer* and is a simplified version of Lafortune's diagnoser [11], *e.g.,* without documenting the faults that occurred.

2.2 How about distributed systems?

Recall the product of automata (also called "parallel composition" in DES literature):

$$\mathcal{A}_1 \times \mathcal{A}_2 \quad = \quad (S_1 \times S_2, L_1 \cup L_2, \rightarrow, (s_{0,1}, s_{0,2})) \tag{2}$$

where $(s_1, s_2) \xrightarrow{\ell} (s_1', s_2')$ iff the automata progress either locally (cases (i) and (iii)) or jointly (case (ii)):

$$\begin{array}{lccccc} \text{(i)} & \ell \in L_1 \setminus L_2 & \wedge & s_1 \xrightarrow{\ell} s_1' & \wedge & s_2' = s_2 \\ \text{(ii)} & \ell \in L_2 \cap L_1 & \wedge & s_1 \xrightarrow{\ell} s_1' & \wedge & s_2 \xrightarrow{\ell} s_2' \\ \text{(iii)} & \ell \in L_2 \setminus L_1 & \wedge & s_1' = s_1 & \wedge & s_2 \xrightarrow{\ell} s_2' \end{array}$$

Next, following our requirements, assume that the automaton for monitoring decomposes as

$$\begin{array}{rcl} \mathcal{A} & = & \times_{i \in I} \mathcal{A}_i \\ \text{where } \mathcal{A}_i & = & (S_i, L_i, \rightarrow_i, s_{i,0}),\ L_i = L_{o,i} \cup L_{u,i}, \\ \text{and } L & = & L_o \cup L_u, \text{ with } L_o = \bigcup_{i \in I} L_{o,i}. \end{array} \tag{3}$$

Do we have:

$$\mathcal{A}_{/L_u} \quad \stackrel{?}{=} \quad \times_{i \in I} \mathcal{A}_{i/L_{u,i}}$$

The answer is yes if there is no hidden interaction:

$$\forall i, j \quad : \quad L_i \cap L_j \cap L_u = \emptyset$$

With this last assumption, computing the monitor can be done entirely locally. However, this is too strong an assumption, as the interesting case is when hidden interactions occur between components, e.g., fault effect propagation. Without this unrealistic assumption, we do not know how to compute the monitor in a distributed way.

2.3 The language based approach

This approach was popularized by Su and Wonham [29, 30]. The idea is the following: eq. (1) can be solved provided $\Sigma_{\mathcal{A}}$ and O are of reasonable size—the above authors call $\Sigma_{\mathcal{A}}$ the *language* of the system. Taking this computation as an atomic step, one can address

large systems in the following manner. Assuming $\mathcal{A}$ decomposes as in (3), we can decompose $\Sigma_{\mathcal{A}}$ as

$$\Sigma_{\mathcal{A}} \quad = \quad \|_{i\in I}\Sigma_{\mathcal{A}_i}$$

where $\|$ denotes the parallel product of languages. Given an observed sequence O, or better, a collection of observed sequences $(O_i)_{i\in I}$, one per component $\mathcal{A}_i$, the monitor is obtained in two steps:

1. compute the set $\mathcal{V}_i = \mathbf{Proj}_{o,i}^{-1}(O_i)$ of all runs of $\mathcal{A}_i$ matching local observations O_i;
2. compute the parallel product of these local sets of runs $\mathcal{V} = \|_{i\in I}\mathcal{V}_i$.

In practice, one is not so much interested in $\mathcal{V}$, *i.e.* runs of $\mathcal{A}$ explaining all observations, than in the projections of $\mathcal{V}$ on each component $\mathcal{A}_i$. As a matter of fact, the latter can be computed more efficiently than $\mathcal{V}$ by a combination of projection, merge and product operators. This approach does not require that the interactions between components are observed. The authors distinguish *local consistency* where the local solutions agree on their interfaces, and *global consistency* where the local solutions are projections of the global solution.

We do not give more details here since the precise description of this approach is detailed in the next section, with a major difference however: instead of assuming that (1) can be effectively solved (which holds for small systems only), we pay attention to the efficiency of the data structure to encode runs of $\mathcal{A}$, and we solve (1) in a distributed way and recursively as the number of observations increases.

3 Unfolding based monitoring

From now on, we shall consider on-line algorithms, which do not pre-compute monitors or observers. Key to this are data structures and techniques to manipulate sets of runs in an efficient way. The simplest one is presented and studied first.

3.1 Unfoldings to represent sets of runs for automata

The principle of unfoldings is to represent sets of runs by superimposing common prefixes of them, thus obtaining a tree-shaped data structure.[1]

Occurrence nets and Unfoldings. An (S, L)-*occurrence net* is a tree whose branches and nodes are labeled by two finite alphabets denoted by L and S, respectively.

Let $\mathcal{A} = (S, L, \rightarrow, s_0)$ be an automaton. Its *unfolding* $\mathcal{U}_{\mathcal{A}}$ is the unique (S, L)-occurrence net whose branches are all the runs of $\mathcal{A}$, each run being represented only once ($\mathcal{U}_{\mathcal{A}}$ is

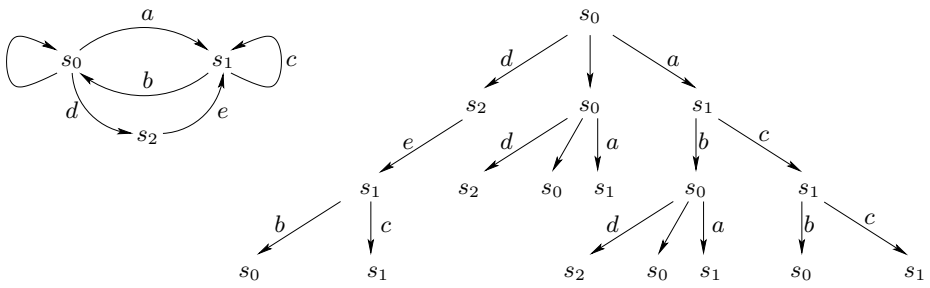

Figure 1: Automaton $\mathcal{A}$ and a prefix of its unfolding $\mathcal{U}_{\mathcal{A}}$.

unique up to an isomorphism of labeled trees). Fig. 1 shows an automaton and a prefix of its unfolding (unfoldings are infinite as soon as automata possess loops).

By abuse, we also call *runs* the maximal branches of any (S, L)-occurrence net. If an automaton $\mathcal{A} = (S, L, \rightarrow, s_0)$ is such that $(S, \rightarrow)$ is a tree, then $\mathcal{U}_{\mathcal{A}}$ identifies with $\mathcal{A}$; by abuse, we say that $\mathcal{A}$ is an unfolding. For example, a single run is an unfolding.

3.2 Unfolding based monitoring

The monitor for $\mathcal{A} = (S, L, \rightarrow, s_0), L = L_o \cup L_u$ is redefined in terms of unfoldings as follows:

$$\mathcal{D} \quad =_{\text{def}} \quad \mathcal{U}_{\mathcal{A}\times O} \tag{4}$$

The so constructed $\mathcal{D}$ contains all runs of $\mathcal{D}$ that explain some prefix of O. We can recover our original definition (1) from $\mathcal{D}$ by pruning away the runs of $\mathcal{D}$ that do not explain O entirely.

This is illustrated in Fig. 2. The construction of $\mathcal{D}$ can be performed incrementally and on-line, while successive events of O are received. Note that the branch $(s_1, 0) \xrightarrow{f} (s_3, 1)$ belonging to $\mathcal{D}$ fails to explain the postfix $\{r, f\}$ of the observation sequence. It should therefore be pruned to get the desired form (1). Such a pruning can be performed with delay exactly 1, *i.e.,* when receiving the event labeled r in O.

In fact, this definition also works if O is not an observation sequence, but rather an automaton. For example, $O = O_1 \times O_2$ can be the automaton representing the set of interleavings of two concurrent measurements by two independent and non synchronized sensors. This means that we can as well consider the case of distributed sensing.

[1] Unfoldings associated to automata, *i.e.,* sequential machines, are usually called *execution trees.* However, since we shall consider both sequential and partial order techniques, we prefer calling them already unfoldings.

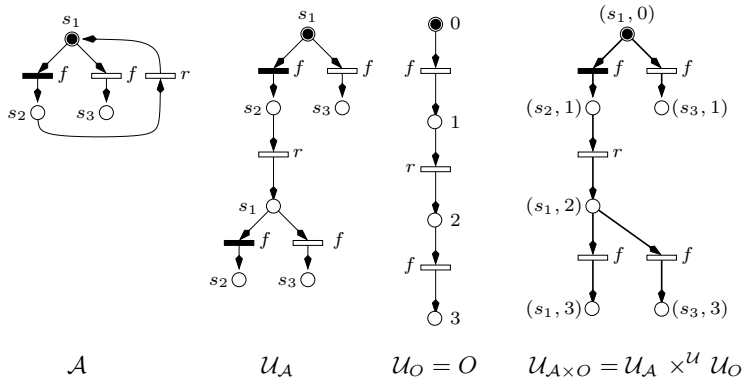

Figure 2: Computing $\mathcal{D} = \mathcal{U}_{\mathcal{A}\times O}$.

3.3 Basic operators on occurrence nets and unfoldings

To develop our distributed on-line algorithms, we will need the following operators on unfoldings or occurrence nets: intersections, projections, and products. These are introduced next.

Intersection. For $\mathcal{V}$ and $\mathcal{V}'$ two (S, L)-occurrence nets, their *intersection*

$$\mathcal{V} \cap \mathcal{V}'$$

is the (S, L)-occurrence net whose runs are the common runs of $\mathcal{V}$ and $\mathcal{V}'$.

Projection. Let $\mathcal{V}$ be an (S, L)-occurrence net. For $L' \subseteq L$ and $\pi : S \mapsto S'$ a total surjection from S onto some alphabet S', let

$$\mathbf{Proj}_{L',\pi}(\mathcal{V}) \tag{5}$$

be the projection of $\mathcal{V}$ on L', obtained by applying the following two rules, where the term "maximal" refers to partial ordering by inclusion:

1. any maximal branch

$$s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} s_2 \xrightarrow{\ell_3} s_3 \ldots s_{n-1} \xrightarrow{\ell_n} s_n$$

   such that $\forall k = 1, \ldots, n-1, \ell_k \notin L'$ and $\ell_n \in L'$, is replaced by $\pi(s_0) \xrightarrow{\ell_n} \pi(s_n)$;

2. any maximal branch

$$s_0 \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} s_2 \xrightarrow{\ell_3} s_3 \ldots s_{n-1} \xrightarrow{\ell_n} s_n$$

such that $\forall k = 1, \ldots, n, \ell_k \notin L'$, is replaced by $\pi(s_0)$.

States that are not connected are removed. If $\mathcal{V}$ is a prefix of the unfolding $\mathcal{U}_\mathcal{A}$ of some product automaton $\mathcal{A} = \mathcal{A}' \times \mathcal{A}''$, then we simply write $\mathbf{Proj}_{\mathcal{A}'}(\mathcal{V})$ instead of $\mathbf{Proj}_{L',\pi}(\mathcal{V})$.

Product. Using the self-reproducing property $\mathcal{U}_\mathcal{V} = \mathcal{V}$ if $\mathcal{V}$ is an occurrence net, we can define a notion of product for occurrence nets as follows:

$$\mathcal{V} \times^\mathcal{U} \mathcal{V}' \quad =_{\text{def}} \quad \mathcal{U}_{\mathcal{V} \times \mathcal{V}'}$$

where the product of automata was defined in (2). Such products are associative and commutative. The runs of $\mathcal{V} \times^\mathcal{U} \mathcal{V}'$ are simply obtained by synchronizing the runs of $\mathcal{V}$ and of $\mathcal{V}'$.

3.4 Factorizing unfoldings

The following result is instrumental in getting the distributed monitoring algorithms:

Theorem 1 ([14]) *We are given a product $\mathcal{A} = \times_{i \in I} \mathcal{A}_i$ of automata.*

1. *We have*

$$\mathcal{U}_\mathcal{A} \quad = \quad \times^\mathcal{U}_{i \in I} \mathcal{U}_{\mathcal{A}_i} \quad = \quad \times^\mathcal{U}_{i \in I} \mathbf{Proj}_{\mathcal{A}_i}(\mathcal{U}_\mathcal{A})$$

   where each $\mathbf{Proj}_{\mathcal{A}_i}(\mathcal{U}_\mathcal{A})$ is a (generally strict) prefix of $\mathcal{U}_{\mathcal{A}_i}$.

2. *For each $i \in I$, let $\mathcal{V}_i$ be a prefix of the unfolding $\mathcal{U}_{\mathcal{A}_i}$, and let $\mathcal{V} =_{\text{def}} \times^\mathcal{U}_{i \in I} \mathcal{V}_i$ be their product. Then $\mathcal{V}$ is a prefix of $\mathcal{U}_\mathcal{A}$,* i.e. *a valid set of runs for $\mathcal{A}$, and as above one has*

$$\mathcal{V} \quad = \quad \times^\mathcal{U}_{i \in I} \mathcal{V}_i \quad = \quad \times^\mathcal{U}_{i \in I} \mathbf{Proj}_{\mathcal{A}_i}(\mathcal{V})$$

   where each $\mathcal{V}_i^* =_{\text{def}} \mathbf{Proj}_{\mathcal{A}_i}(\mathcal{V})$ is a (generally strict) prefix of $\mathcal{V}_i$.

   In addition, $\mathcal{V}_i^*$ is the minimal decomposition of $\mathcal{V}$ according to alphabets L_i in that any other decomposition $\mathcal{V} = \times^\mathcal{U}_{i \in I} \mathcal{V}'_i$, where $\mathcal{V}'_i$ has alphabet L_i, is such that $\mathcal{V}_i^*$ is a prefix of $\mathcal{V}'_i$.

This theorem says that the unfolding of $\mathcal{A}$ can be computed as a product of unfoldings, and point 2 expresses that any set of runs defined by a product form actually admits a minimal product form, defined by its projections. Theorem 1 is a fundamental result to develop distributed algorithms based on unfoldings.

3.5 Unfolding based modular monitoring

Now, we consider a distributed setting in which both the system and its sensors are distributed:

$$\begin{array}{rcl} \mathcal{A} = & \times_{i\in I}\mathcal{A}_i & , \quad L_i = L_{i,o} \cup L_{i,u} \\ O = & \times_{i\in I} O_i & , \quad \text{with alphabet } L_{i,o} \end{array}$$

We allow that not all interactions are observed, *i.e.*

$$L_i \cap L_j \not\subseteq L_{i,o} \cap L_{j,o} \text{ is allowed,}$$

and we also allow that pairs of interacting components disagree on which label is observed or unobserved, *i.e.*

$$L_{i,o} \cap L_j \neq L_{j,o} \cap L_i \text{ is allowed.}$$

Using Theorem 1 in (4) yields:

$$\mathcal{D} \quad = \quad \times^{\mathcal{U}}_{i\in I} \mathcal{U}_{\mathcal{A}_i \times O_i} \quad = \quad \times^{\mathcal{U}}_{i\in I} \left(\mathcal{U}_{\mathcal{A}_i} \times^{\mathcal{U}} O_i\right)$$

This suggests defining *modular monitoring* by

$$\mathcal{D}_{\text{mod}} =_{\text{def}} (\mathcal{D}_i)_{i\in I}\,, \qquad \text{where } \mathcal{D}_i = \mathbf{Proj}_{\mathcal{A}_i \times O_i}(\mathcal{D}) \tag{6}$$

and the latter satisfies $\times^{\mathcal{U}}_{i\in I} \mathcal{D}_i = \mathcal{D}$.

3.6 Six basic problems

The following basic problems must be addressed, we shall do this in the sequel:

Problem 1 *Compute $\mathcal{D}_{\text{mod}}$ without computing $\mathcal{D}$.*

Problem 2 *Compute $\mathcal{D}_{\text{mod}}$ by attaching a supervising peer to each site.*

Problem 3 *Compute $\mathcal{D}_{\text{mod}}$ on-line and on the fly.*

Problem 4 *Address asynchronous distributed systems.*

Problem 5 *Avoid state explosion due to the concurrency between and possibly within the different components.*

Problem 6 *Address changes in the systems dynamics.*

3.7 Distributed monitoring—Problems 1 and 2

Distributed monitoring relies on the following fundamental result:

Theorem 2 ([14, 15, 19]) *Let $(\mathcal{A}_i)_{i=1,2,3}$ be three automata such that*

$$(L_1 \cap L_3) \subseteq L_2 \qquad (\mathcal{A}_2 \text{ separates } \mathcal{A}_1 \text{ from } \mathcal{A}_3)$$

and consider a prefix of $\mathcal{U}_{\mathcal{A}_1 \times \mathcal{A}_2 \times \mathcal{A}_3}$ defined by $\mathcal{V}_1 \times^{\mathcal{U}} \mathcal{V}_2 \times^{\mathcal{U}} \mathcal{V}_3$, where $\mathcal{V}_i$ is some prefix of $\mathcal{U}_{\mathcal{A}_i}$. Write $\mathbf{Proj}_i(.)$ for short instead of $\mathbf{Proj}_{\mathcal{A}_i}(.)$. Then, the following formulas hold:

$$\mathbf{Proj}_2(\mathcal{V}_1 \times^{\mathcal{U}} \mathcal{V}_2 \times^{\mathcal{U}} \mathcal{V}_3) = \underbrace{\underbrace{\mathbf{Proj}_2(\mathcal{V}_1 \times^{\mathcal{U}} \mathcal{V}_2)}_{\text{local to (1,2)}} \bigcap \underbrace{\mathbf{Proj}_2(\mathcal{V}_2 \times^{\mathcal{U}} \mathcal{V}_3)}_{\text{local to (2,3)}}}_{\text{local to 2}} \tag{7}$$

$$\mathbf{Proj}_1(\mathcal{V}_1 \times^{\mathcal{U}} \mathcal{V}_2 \times^{\mathcal{U}} \mathcal{V}_3) = \underbrace{\mathbf{Proj}_1(\mathcal{V}_1 \times^{\mathcal{U}} \underbrace{\mathbf{Proj}_2(\mathcal{V}_2 \times^{\mathcal{U}} \mathcal{V}_3)}_{\text{local to (2,3)}})}_{\text{local to (1,2)}} \tag{8}$$

Proof: Easy if we remember that the runs of $\mathcal{V} \times^{\mathcal{U}} \mathcal{V}'$ are obtained by synchronizing the runs of $\mathcal{V}$ and the runs of $\mathcal{V}'$. ◇

Define the following operators, attached to the pair of sites (i, j) and site i, respectively:

$$\begin{aligned} \mathbf{Msg}_{\mathcal{V}_i \to \mathcal{V}_j} &=_{\text{def}} \mathbf{Proj}_j(\mathcal{V}_j \times^{\mathcal{U}} \mathcal{V}_i) \\ \mathbf{Fuse}(\mathcal{V}_i, \mathcal{V}'_i) &=_{\text{def}} \mathcal{V}_i \cap \mathcal{V}'_i \end{aligned}$$

Notice that the **Fuse** operator generalizes to any number of messages. Using these operators, rules (7) and (8) respectively rewrite as

$$\mathbf{Proj}_2(\mathcal{V}_1 \times^{\mathcal{U}} \mathcal{V}_2 \times^{\mathcal{U}} \mathcal{V}_3) = \mathbf{Fuse}(\mathbf{Msg}_{\mathcal{V}_1 \to \mathcal{V}_2}, \mathbf{Msg}_{\mathcal{V}_3 \to \mathcal{V}_2}) \tag{9}$$

$$\mathbf{Proj}_1(\mathcal{V}_1 \times^{\mathcal{U}} \mathcal{V}_2 \times^{\mathcal{U}} \mathcal{V}_3) = \mathbf{Msg}_{(\mathbf{Msg}_{\mathcal{V}_3 \to \mathcal{V}_2}) \to \mathcal{V}_1} \tag{10}$$

Let $(\mathcal{A}_i)_{i \in I}$ be a collection of automata. Define its *interaction graph* as the following non directed graph: its vertices are labeled with the indices $i \in I$, and we draw a branch (i, j) iff no other index $k \in I$ exist such that $\mathcal{A}_k$ separates $\mathcal{A}_i$ from $\mathcal{A}_j$.

Fig. 3 illustrates the resulting *belief propagation* algorithm when the interaction graph of $(\mathcal{A}_i)_{i \in I}$ is a tree. This algorithm results from successive applications of Thm. 2 with the scheduling indicated by the index from 1 (1st step) to 6 (last step). The arrows depict message propagation, and fusion occurs when two or more messages reach the same node. At the end, a fusion of all incoming messages is performed at each node, which yields the desired projection of $\times^{\mathcal{U}}_{i \in I} \mathcal{V}_i$ on each node.

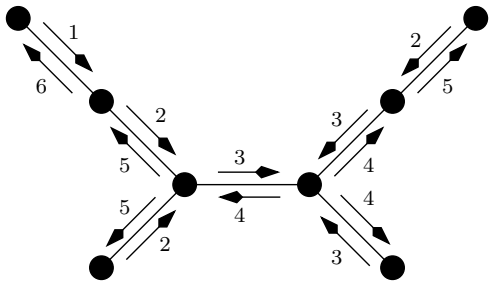

Figure 3: Belief propagation algorithm when the interaction graph of $(\mathcal{A}_i)_{i\in I}$ is a tree.

For the application to distributed monitoring, simply perform the substitution: $\mathcal{A}_i \leftarrow \mathcal{A}_i \times O_i$. Note that interactions between components may occur through unobserved labels (this is in fact the interesting case for fault diagnosis).

The above rigid and strongly synchronized scheduling is not acceptable for distributed monitoring. The following lemma helps overcoming this:

Lemma 1 ([17]) *The two maps*

$$\begin{array}{rcl}(\mathcal{V}_i, \mathcal{V}_j) & \mapsto & \mathbf{Msg}_{\mathcal{V}_i \to \mathcal{V}_j} \\ (\mathcal{V}_i, \mathcal{V}_i') & \mapsto & \mathbf{Fuse}(\,\mathcal{V}_i\,,\,\mathcal{V}_i'\,)\end{array}$$

are increasing w.r.t. each component.

As a consequence, *chaotic iterations* where messages are sent asynchronously to neighbors, put into buffers at reception, then read and fused at any time to prepare a next message, will converge to the same result as the rigid scheme of Fig. 3. The latter is just the scheme minimizing the number of communications between sites.

When the interaction graph of $(\mathcal{A}_i)_{i\in I}$ possesses cycles, then this algorithm can still be used. At the equilibrium, it yields *local consistency* in the sense of [28, 29, 30], meaning that local monitors agree on their interfaces. However this algorithm does not compute in general local projections of the global monitor $\mathbf{Proj}_i(\mathcal{D})$, it only computes some upper approximation of them, see [16].

So far this addressed Problems 1 and 2. Next, we consider Problem 3.

3.8 Distributed on-line monitoring—Problem 3

We shall see that solving the latter can be done again by using Lemma 1. To derive on-the-fly belief propagation, consider the following additional operator attached to site i:

$$\mathbf{Grow}(\,O_i\,,\,\ell_i\,) \quad =_{\text{def}} \quad \text{append to } O_i \text{ a new local event labeled } \ell_i \tag{11}$$

and consider also the following atomic operator obtained as follows: pick a neighboring node i_0 of i, denote by $i_1, \ldots, i_n$ the other neighboring nodes of i, and perform:

$$\mathcal{V}_i := \mathbf{Fuse}(\, \mathcal{V}_{i_1} \,,\, \mathcal{V}_{i_2} \,, \ldots,\, \mathcal{V}_{i_n} \,)\,;\; \mathbf{Msg}_{\,\mathcal{V}_i \to \mathcal{V}_{i_0}} \tag{12}$$

Each site performs one of the two operations (11) or (12), asynchronously, in a chaotic way. Thanks to Lemma 1, the resulting chaotic iterations converges to the same value as for the scheme shown in Fig. 3, and the algorithm is incremental. See [17] for a detailed analysis in a partial order context.

4 Trellis based monitoring—Problems 4 and 5

So far we seem to have reached a satisfactory solution of Problems 1 – 3. Did we address Problems 4 and 5? Not quite so: our solution is somehow cheating. In general, unfoldings grow exponentially in width with their length, see Fig. 1. This becomes prohibitive when considering on-the-fly algorithms. We would be happy with data structures having bounded width along the processing. Trellises, which have been used for a long time in dynamic programming algorithms, are good candidates for this.

In this section we discuss trellis based monitoring. Again, we play the same game by first insisting that nothing fancy shall be introduced. So we stick with the classical sequential setting (automata and their products). At some point, however, we will see that considering partial orders cannot be avoided.

4.1 Observation criteria and trellises

Unfoldings are a simple structure to represent sets of runs, for automata. However, when a path of the unfolding branches, its descendants separate for ever. *Trellises* have been used in dynamic programming (or in the popular Viterbi algorithm), by merging, in the unfolding, futures of different runs according to appropriate criteria. For example, merge the final nodes of two finite runs σ and σ' if:

1. They begin and terminate at identical states (this first condition is mandatory to ensure that σ and σ' have identical futures);
2. They are equivalent according to one of the following *observation criteria*:
   (a) σ and σ' possess identical length;[2]
   (b) σ and σ' possess identical visible length (by not counting silent transitions);
   (c) Select some $L_o \subset L$ and require that σ and σ' satisfy $\mathbf{Proj}_{L_o}(\sigma) = \mathbf{Proj}_{L_o}(\sigma')$;
   (d) Assume $\mathcal{A} = \times_{i \in I} \mathcal{A}_i$ and require that σ and σ' have identical lengths when restricted to the different local alphabets L_i.

[2] This is the observation criterion used in dynamic programming or Viterbi algorithm.

We now formalize the concept of observation criterion:

Definition 1 (observation criterion) *An* observation criterion $\theta : L \mapsto L_\theta$ *is a partial function relating two finite alphabets;* θ *extends to words as usual, and we take the convention that* $\theta(w) = \epsilon$*, the empty word, if no symbol of* w *has an image via* θ*.*

Let $\mathcal{T}$ be a graph whose nodes are labeled by S and branches are labeled by $L \cup \{\star\}$ (where special symbol $\star$ means the absence of label). For θ an observation criterion, say that two branches

$$s_{\text{init}} \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} s_2 \xrightarrow{\ell_3} s_3 \dots s_{n-1} \xrightarrow{\ell_n} s_{\text{end}}$$

and

$$s'_{\text{init}} \xrightarrow{\ell'_1} s'_1 \xrightarrow{\ell'_2} s'_2 \xrightarrow{\ell'_3} s'_3 \dots s'_{m-1} \xrightarrow{\ell'_m} s'_{\text{end}}$$

of $\mathcal{T}$ are θ-equivalent iff

$$\begin{aligned} & s_{\text{init}} = s'_{\text{init}}, s_{\text{end}} = s'_{\text{end}} \\ \text{and} \quad & \theta(\ell_1\ell_2\ell_3 \dots \ell_n) = \theta(\ell'_1\ell'_2\ell'_3 \dots \ell'_m) \end{aligned}$$

Notation. By abuse of notation, we shall sometimes write $\theta(\sigma)$ instead of $\theta(\ell_1\ell_2\ell_3 \dots \ell_n)$, when $\ell_1\ell_2\ell_3 \dots \ell_n$ is the word produced by run σ, as above.

Definition 2 *Let* $\mathcal{T}$ *be a directed graph whose nodes are labeled by* S *and branches are labeled by* $L \cup \{\star\}$*, and let* $\theta : L \mapsto L_\theta$ *be an observation criterion.* $\mathcal{T}$ *is an* (S, L, θ)-trellis *if it satisfies the following condition: any two branches originate from the same node of* $\mathcal{T}$ *and terminate at the same node of* $\mathcal{T}$ *iff they are* θ*-equivalent.*

As a consequence, every circuit of $\mathcal{T}$ must be labeled by a word whose image by θ is ϵ. Examples corresponding to the above cases (a)–(d) are

(a) $L_\theta = \{1\}$, $\text{Dom}(\theta) = L \cup \{\star\}$.

(b) $L_\theta = \{1\}$, $\text{Dom}(\theta) = L$.

(c) $L_\theta = L_o$, and $\theta(\ell) = \ell$ iff $\ell \in L_o$, $\theta(\ell)$ being otherwise undefined.

(d) $L_\theta = I$, and $\theta(\ell) = i$ if $\ell \in L_i$.

For $\mathcal{V}$ an (S, L)-occurrence net and $\theta : L \mapsto L_\theta$ an observation criterion, the pair $(\mathcal{V}, \theta)$ gives raise to a *trellis* $\mathcal{T}(\mathcal{V}, \theta)$, obtained by merging extremal states of minimal (for inclusion) θ-equivalent branches of $\mathcal{V}$. For $\mathcal{A} = (S, L, \rightarrow, s_0)$ an automaton, and θ an observation criterion, define

$$\mathcal{T}_{\mathcal{A},\theta} \quad =_{\text{def}} \quad \mathcal{T}(\mathcal{U}_\mathcal{A}, \theta)$$

Trellises are illustrated in Fig. 4, for the above cases (a), (b), and (c). Case (d) will be discussed later.

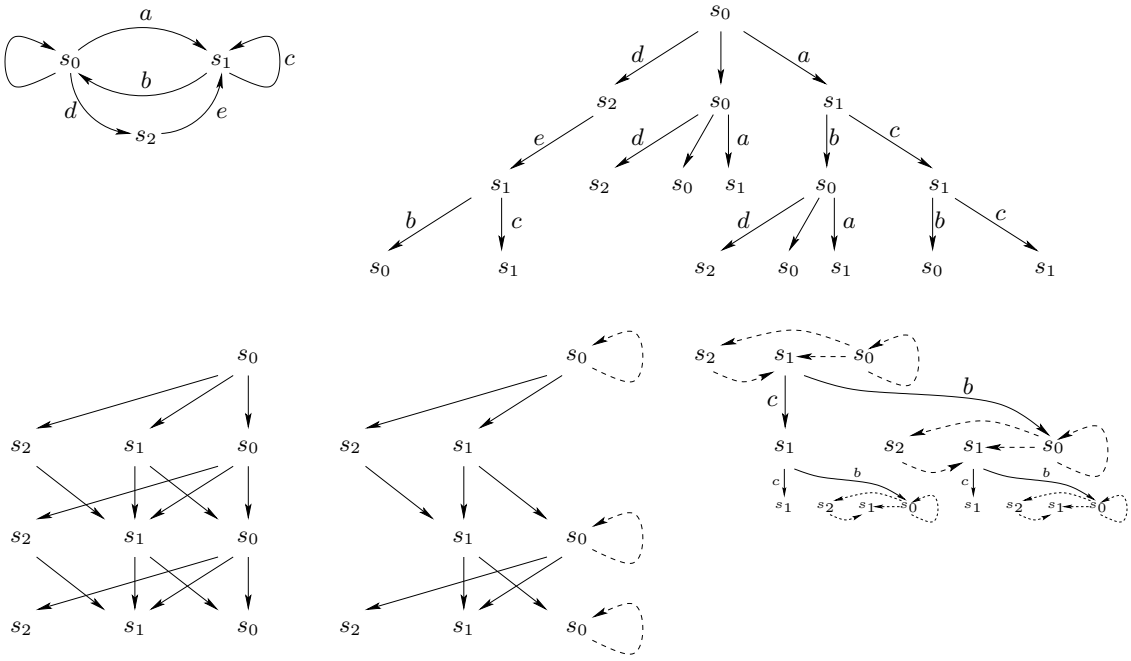

Figure 4: Top. Left: $\mathcal{A}$; right: unfolding $\mathcal{U}_{\mathcal{A}}$. Bottom. Left: $\mathcal{T}_{\mathcal{A}}^{(a)}$; mid: $\mathcal{T}_{\mathcal{A}}^{(b)}$; right: $\mathcal{T}_{\mathcal{A}}^{(c)}$, with $L_o = \{b, c\}$. Labels of transitions are omitted in the trellises. Loops in trellises are dashed, they correspond to paths in the unfolding whose labels are undefined under θ.

4.2 Trellis based monitors

The trellis based monitor for $\mathcal{A} = (S, L, \rightarrow, s_0), L = L_o \cup L_u$ is defined as

$$\mathcal{D} \quad =_{\text{def}} \quad \mathcal{T}_{(\mathcal{A}\times O),\theta} \tag{13}$$

where the observation criterion θ is discussed next. Consider the following three alternatives for θ:

(i) Observation criterion $\theta : L \mapsto \{1\}$ is the partial function such that $\theta(\ell) = 1$ if $\ell \in L_o$, and otherwise $\theta(\ell)$ is undefined—observation criterion θ counts the visible global length;

(ii) Observation criterion $\theta : L \mapsto L_o$ is the partial function such that $\theta(\ell) = \ell$ if $\ell \in L_o$, and otherwise $\theta(\ell)$ is undefined—observation criterion θ records the global observed sequence;

(iii) For $\mathcal{A} = \times_{i\in I} \mathcal{A}_i$ and $O = \times_{i\in I} O_i$, we also consider the observation criterion $\theta : L \mapsto I$, which is the partial function such that $\theta(\ell) = i$ if $\ell \in L_{o,i}$, and otherwise

undefined—observation criterion θ counts the visible local lengths. (This corresponds to case (d) of previous section.)

Note that observation criterion (i) is the classical one, used in dynamic programming. It was illustrated by diagram $\mathcal{T}_{\mathcal{A}}^{(a)}$ of Fig. 4. Also, note that $\mathcal{D}$ *as defined in (13) can be computed on-line along with the recording of the observation* O.

Comparing the above three observation criteria. Let $(O_i)_{i\in I}$ be a tuple of local observation sequences collected by the different sensors. Then, $O = \times_{i\in I} O_i$, their product, is in fact the set of all possible interleavings of the local observations O_i. Then, every run of $\mathcal{D}$ explains some prefix of one among those interleavings.

Two such runs, σ and σ', will be merged according to observation criterion (i) iff 1/ they terminate at identical states of $\mathcal{A} \times O$ and 2/ they possess identical global length. In fact, the terminal state of σ (or σ') contains, as part of its components, the terminal state of its O component, which is a tuple $(n_i)_{i\in I}$, where n_i is the length of observation O_i (see Fig. 2 for the coding of observations). Thus having identical terminal states implies, for σ and σ', that they have explained observations with equal local lengths. Thus, although observation criteria (i) and (ii) differ for general trellises, they coincide for the particular trellises $\mathcal{T}_{(\mathcal{A}\times O),\theta}$ defining monitors, because of the presence and special form of O.

On the other hand, since each local observation consists of a single sequence, knowing the length of a prefix of it entirely determines this prefix. Therefore, observation criteria (ii) and (iii) are again equivalent for use in monitoring.

To summarize, observation criteria (i), (ii), and (iii) differ in general, but they are equivalent when used in the context of monitoring, *i.e.,* they will result in identical merges. $\diamond$

In the next section we will see that observation criteria (ii) and (iii) yield valid calculi involving intersections, projections, and products, whereas (i) won't. And we will explain why.

4.3 Basic operators on trellises

Basic operators are defined next.

Intersection. For $\mathcal{T}$ and $\mathcal{T}'$ two (S, L, θ)-trellises, their *intersection* $\mathcal{T} \cap \mathcal{T}'$ is the unique (S, L, θ)-trellis whose runs are the common runs of $\mathcal{T}$ and $\mathcal{T}'$.

Products. Two observation criteria $\theta : L \mapsto L_\theta$ and $\theta' : L' \mapsto L_{\theta'}$ are called *compatible* if θ and θ' agree on $L \cap L'$; in this case, define their *join* $\theta \sqcup \theta'$ by

$$(\theta \sqcup \theta')(\ell) \quad = \quad \text{if } \ell \in L \text{ then } \theta(\ell) \text{ else } \theta'(\ell)$$

Assuming θ and θ' compatible, define

$$\mathcal{S} \times^{\mathcal{T}} \mathcal{S}' \quad =_{\text{def}} \quad \mathcal{T}_{\mathcal{U}_{\mathcal{S}\times\mathcal{S}'}} \ (= \mathcal{T}_{\mathcal{U}_{\mathcal{S}} \times^{\mathcal{U}} \mathcal{U}_{\mathcal{S}'}}) \tag{14}$$

where the observation criterion used in defining $\mathcal{T}_{\mathcal{U}_{S\times S'}}$ is $\theta\sqcup\theta'$. Such products are associative and commutative.

Projection. Projections can be defined in the same way for trellises as for unfoldings. Let $\mathcal{T}$ be an (S, L, θ)-trellis, and let $L' \subseteq L$ and $\pi : S \mapsto S'$ a total surjection from S onto some alphabet S'. Define the projection $\mathbf{Proj}_{L',\pi}(\mathcal{T})$ as in (5) by applying rules 1 and 2 to the branches of $\mathcal{T}$.

4.4 Problems with some observation criteria

The above notions raise a number of difficulties, depending on the observation criteria used. The following two problems occur when using θ as in case (i) above.

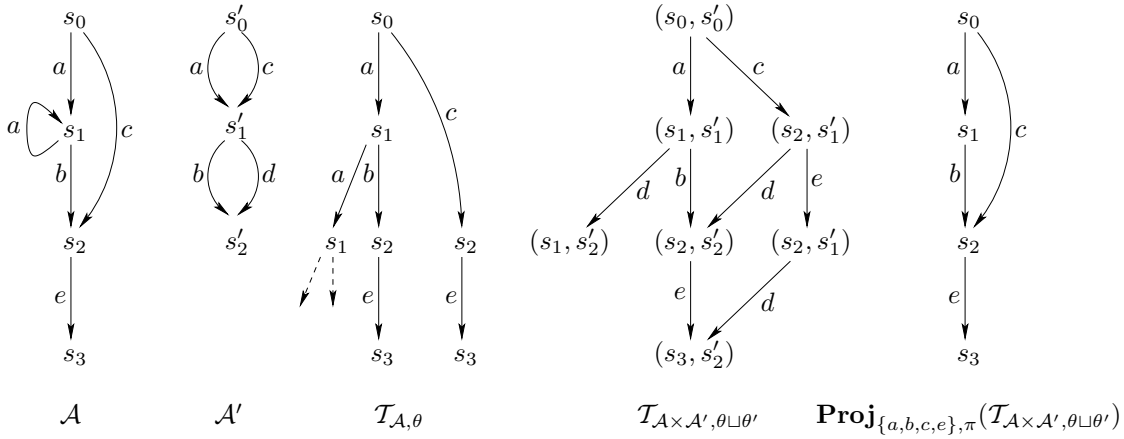

Figure 5: Illustrating a problem with products and projections of trellis. Observation criterion is by counting the number of non-silent branches leading to the considered event. The projection consists in 1/ erasing the events not labeled by a, b, c, e, and 2/ removing via projection π the primed component of the state.

The trellis structure is not stable under projections if θ counts the visible length, globally (figure 5). The last diagram shown is obtained by performing projection as explained. It does not yield a valid trellis, however, since the two branches $s_0 \xrightarrow{a} s_1 \xrightarrow{b} s_2$ and $s_0 \xrightarrow{c} s_2$ shoud not be confluent because they have different lengths.[3]

[3]We may insist living with this problem and still use such trellises with their products and projections; unfortunately, correcting this may require unbounded backtracking of $\mathbf{Proj}_{\{a,b,c,e\},\pi}(\mathcal{T}_{\mathcal{A}\times\mathcal{A}',\theta\sqcup\theta'})$ in order to remove incorrect merges.

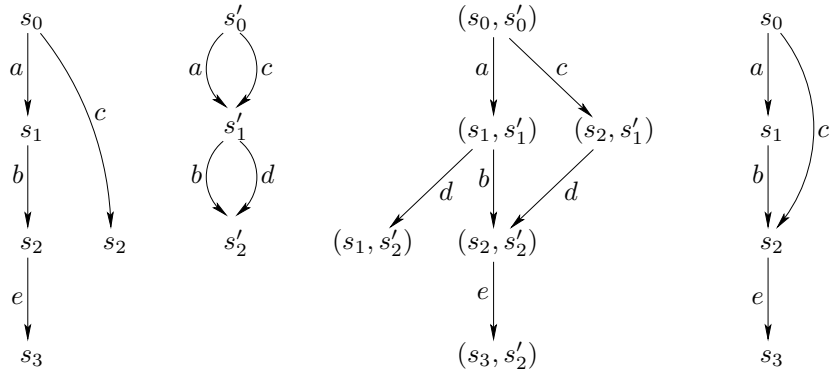

Figure 6: A problem in capturing prefixes of runs. Diagram 1: a prefix of $\mathcal{T}_{\mathcal{A},\theta}$. Diagram 2: $\mathcal{T}_{\mathcal{A}',\theta'}$. Diagram 3: taking the product of diagrams 1 and 2. Diagram 4: projecting this product on the 1st component yields a fake run: $s_0 \xrightarrow{c} s_2 \xrightarrow{e} s_3$ which is not part of the 1st diagram.

Projecting prefixes of trellis yields fake additional runs if θ counts the visible length, globally (figure 6). Usually, when projecting the language of a product automaton, prefixes of runs of the product project into the corresponding prefixes of runs of the components. This is not the case here.

What is the problem? The problem with this global observation criterion is that it is not preserved by projections. This leads us to characterize which are the valid observation criteria to handle distributed systems.

4.5 Revisiting observation criteria and trellises

Reconsider the same problem on the same example of Fig. 5, by using now observation criterion (iii) of Section 4.2. The result is shown on Fig. 7. Why is this the right solution?

The fundamental reason is that $\Theta =_{\text{def}} \theta \sqcup \theta'$ projects well: if $\Theta(\sigma_1) = \Theta(\sigma_2)$ for some pair (σ_1, σ_2) of runs, then we must have $\theta\left(\mathbf{Proj}_{\mathcal{A}}(\sigma_1)\right) = \theta\left(\mathbf{Proj}_{\mathcal{A}}(\sigma_2)\right)$ and $\theta'\left(\mathbf{Proj}_{\mathcal{A}'}(\sigma_1)\right) = \theta'\left(\mathbf{Proj}_{\mathcal{A}'}(\sigma_2)\right)$. We formalize this next by revisiting Definition 1.

Definition 3 (distributable observation criterion) *Let $L = \bigcup_{i\in I} L_i$ be a decomposition of alphabet L, and let $(\theta_i)_{i\in I}$ be a family of pairwise compatible observation criteria. Set $\Theta =_{\text{def}} \bigsqcup_{i\in I} \theta_i$. Say that Θ is* distributable[4] *if, for any two words $w, w' \in L^*$ (the Kleene closure of L):*

$$\Theta(w) = \Theta(w') \quad \Rightarrow \quad \theta_i(\pi_i(w)) = \theta_i(\pi_i(w')) \text{ holds, for every } i \in I,$$

where $\pi_i : L^* \mapsto L_i^*$ is the map consisting in erasing the symbols not belonging to L_i.

[4]Distributable observation criteria are called *height* by E. Fabre in [18].

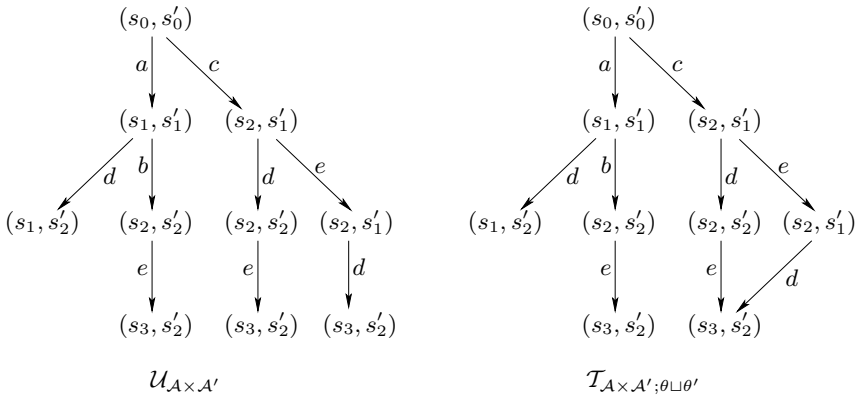

Figure 7: Some new diagrams are shown: $\mathcal{U}_{\mathcal{A}\times\mathcal{A}'}$ is the interleaving based unfolding of $\mathcal{A}\times\mathcal{A}'$; $\mathcal{T}_{\mathcal{A}\times\mathcal{A}';\theta\sqcup\theta'}$ is the interleaving based trellis of $\mathcal{A}\times\mathcal{A}'$, built with observation criterion $\theta\sqcup\theta'$, where θ and θ' count the number of transitions performed by $\mathcal{A}$ and $\mathcal{A}'$, respectively. Note that this observation criterion is made visible here by simply collecting the pairs (i,j) of indices of the compound states (s_i, s'_j) of the product.

The problem with observation criterion (i) of Section 4.2 is that it is not distributable, whereas (ii) and (iii) are distributable. Trellises built with distributable observation criteria can be factorized as shown next.

4.6 Factorizing trellises

Theorem 3 ([19]) *Let $\mathcal{A} = \times_{i\in I}\mathcal{A}_i$ be a product automaton and $\Theta =_{\rm def} \bigsqcup_{i\in I}\theta_i$ be a corresponding* distributable *observation criterion.*

1. *We have*

$$\mathcal{T}_{\mathcal{A},\Theta} \;=\; \times^{\alpha}_{i\in I}\mathcal{T}_{\mathcal{A}_i,\theta_i} \;=\; \times^{\alpha}_{i\in I}\mathbf{Proj}_i(\mathcal{T}_{\mathcal{A},\Theta})$$

   where $\mathbf{Proj}_i()$ denotes the projection on $\mathcal{A}_i$.

2. *For each $i\in I$, let $\mathcal{T}_i$ be a prefix of the trellis unfolding $\mathcal{T}_{\mathcal{A}_i,\theta_i}$, and let $\mathcal{T} =_{\rm def} \times^{\alpha}_{i\in I}\mathcal{T}_i$ be their trellis product. We have*

$$\mathcal{T} \;=\; \times^{\alpha}_{i\in I}\mathbf{Proj}_i(\mathcal{T})$$

   In addition, $\mathcal{T}_i^* =_{\rm def} \mathbf{Proj}_i(\mathcal{T})$ is the minimal decomposition of $\mathcal{T}$ according to alphabets L_i in that any other decomposition $\mathcal{T} = \times^{\alpha}_{i\in I}\mathcal{T}'_i$, where $\mathcal{T}'_i$ has alphabet L_i, is such that $\mathcal{L}_{\mathcal{T}_i^*}\subseteq\mathcal{L}_{\mathcal{T}'_i}$.

This theorem is illustrated on Fig. 8. Now, we have all the needed apparatus for redoing what was done for modular unfolding based monitoring. We do not repeat this.

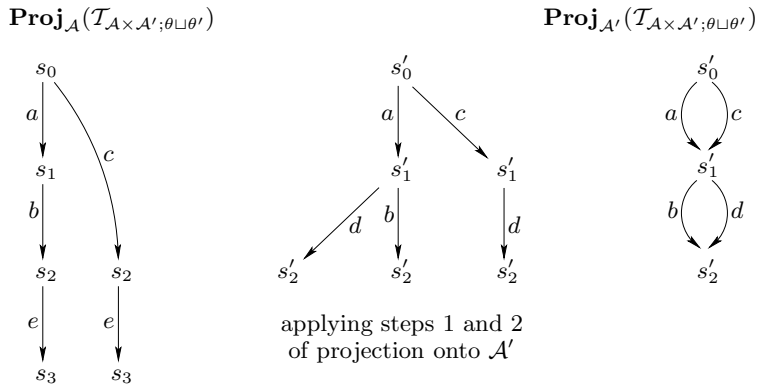

Figure 8: Illustrating Theorem 3 on factorized forms. The mid diagram shows the result of applying rules 1 and 2 defining (5), to the valid trellis $\mathcal{T}_{\mathcal{A}\times\mathcal{A}';\theta\sqcup\theta'}$ shown in Fig. 7. Applying the last step yields the final result.

Discussion. The important property of distributability for an observation criterion should not come as a surprise to us. For example, observation criterion (iii) is nothing but the concept of *vector clock* introduced for the analysis of distributed systems and algorithms in the 80's by Mattern [24] and Fidge [20]. Using vector clocks amounts to regarding executions of the overall distributed system as tuples of synchronized local executions. This is just a partial order view of distributed executions, where local executions are still considered sequential.

5 From trellis to partial order models

In the preceding section, we have seen that runs of distributed systems should be seen as partial orders, obtained by synchronizing the sequential runs of components. Now, if the components of the distributed system interact asynchronously, then internal concurrency also must exist within each component. Hence, the runs of a component should themselves be seen as partial orders. Thus it makes sense to construct a variant of unfoldings or trellises, where runs appear as partial orders. This is illustrated in Figure 9. Advantages and difficulties are discussed next.

Advantages:

- Partial order unfoldings are better than interleaving ones in that they remove diamonds within the component or system considered. This causes reduction in size.
- Furthermore, when long but finite runs are considered for the monitoring problem, it may be that partial order unfoldings perform nearly as well as interleaving based

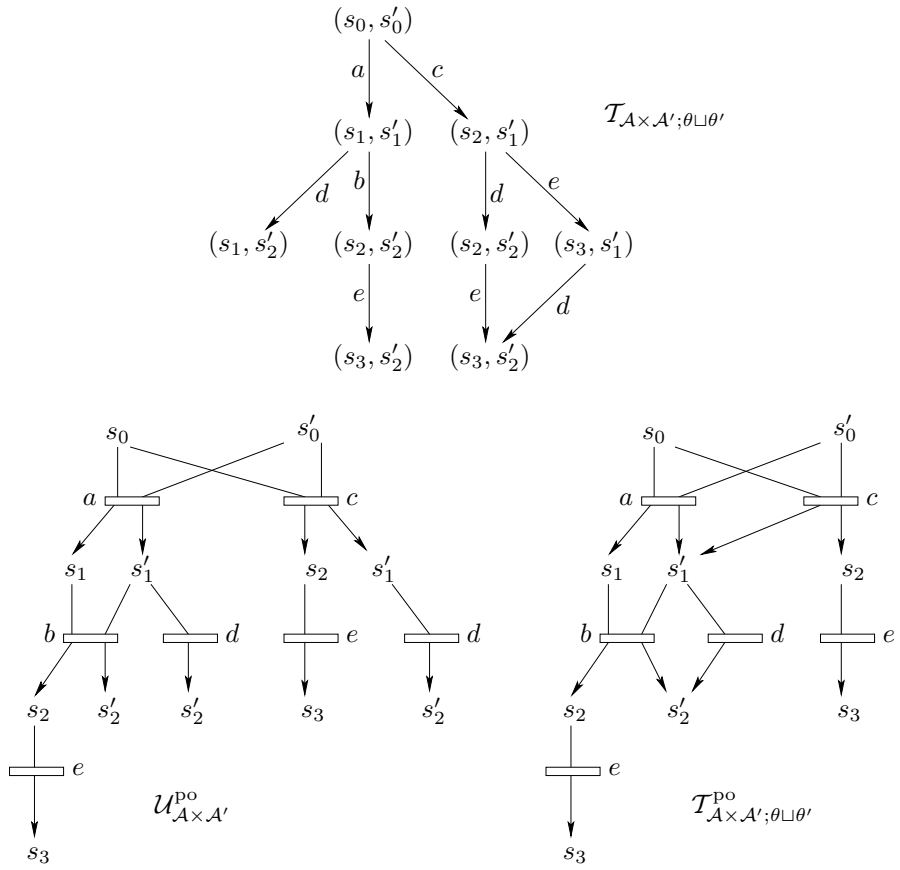

Figure 9: Showing the partial order unfolding $\mathcal{U}^{\mathrm{po}}_{\mathcal{A}\times\mathcal{A}'}$ and trellis $\mathcal{T}^{\mathrm{po}}_{\mathcal{A}\times\mathcal{A}';\theta\sqcup\theta'}$; for comparison, we have left the sequential trellis $\mathcal{T}_{\mathcal{A}\times\mathcal{A}';\theta\sqcup\theta'}$. Note that the diamond has disappeared in both cases.

trellises; this is, *e.g.,* the case when most merge in the considered trellis originate from diamonds in the interleaving semantics.

- Partial order trellises are better than interleaving ones in that they remove diamonds within the component or system considered. This causes reduction in size.
- Partial order unfoldings and trellises can be equipped with notions of product and intersection.

Difficulty: the projection of a partial order unfolding or trellis can sometimes *not* be represented as another partial order unfoldings or trellis, see Figure 10. This figure shows

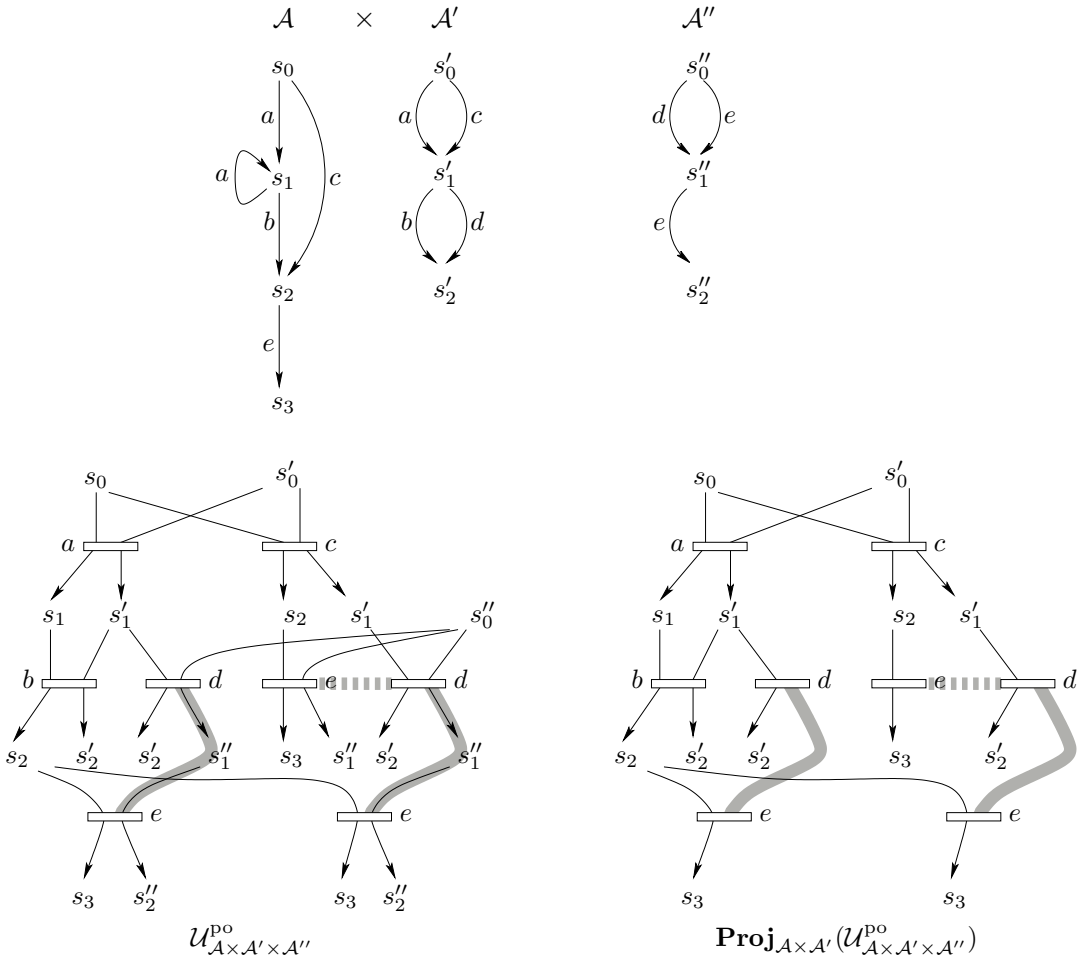

Figure 10: The figure shows a distributed system with two components, written as $(\mathcal{A}\times\mathcal{A}')\times\mathcal{A}''$. This means that the first component is already a distributed system and therefore has internal concurrency. We show on the right the partial order unfolding of this distributed system. Some **conflicts** are depicted in in thick gray dashed lines and some **causalities** are depicted in thick gray solid lines. Projecting on the first component should yield the last diagram, having the conflicts and causalities in it. Unfortunately, these cannot be captured by occurrence net features, with the available nodes. An enriched structure is needed.

the problem with partial order unfoldings, but the same difficulty holds with partial order trellises.

Solutions when using partial order unfoldings. When using partial order unfoldings, the difficulty can be circumvent by one of the following means:

- 1st *method:* enhance occurrence nets with possible additional causalities and conflicts, not resulting from the graph structure of the net. This is the approach taken in [14, 15].
- 2nd *method:* abandon occurrence nets and use *event structures* instead. Event structures are sets of events equipped directly with a causality relation and a conflict relation, with no use of condition nodes to graphically encode conflict. This is the approach taken in [17].
- 3rd *method:* keep occurrence nets as such, but avoid the enhancement used in the 1st method by exchanging messages in the form of so-called *interleaving structures*, see [4].

With these modifications, the preceding techniques for distributed monitoring with partial order unfoldings apply. The development of similar techniques for partial order trellises is under progress.

6 Extensions and further research issues

In this section we review some further problems arising from applications and we draw corresponding research directions.

6.1 Building models for large systems: self-modeling

As explained in Appendix A, realistic applications such as fault management in telecommunication networks and services require models of complexity and size far beyond what can be constructed by hand. Thus, any model based algorithm would fail addressing such type of application unless proper means are found to construct the model.

In some contexts including the one reported in Appendix A, an automatic construction is possible. One approach developed in [1] is called *self-modeling*. Its principle is illlustrated in Fig. 11. To construct models, the following prior information is assumed available:

(a) *A finite set of prototype components is available, and all systems considered are obtained by composing instances of these prototype components.*

In our application context, these prototype components are specified by the different network standards used (as listed in the left most box of Fig. 11), in the form of *Managed Classes*. In this context, the number of classes for consideration is typically small (a dozen or so). In contrast the number of instantiated components in the systems may be huge (from hundreds to thousands).

(b) *For each prototype component, a behavioral model is available in one of the forms we discussed in this paper.*

This is the manual part of the modeling. It was done, *e.g.,* by Alcatel, for the case of all standards shown in the left most box of Fig. 11) [1].

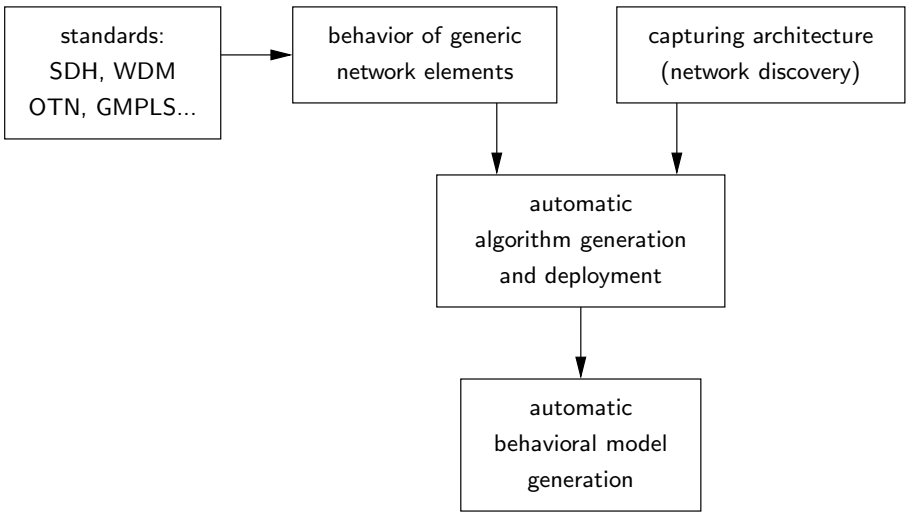

Figure 11: Self-modeling.

(c) *System architecture can be automatically discovered.*

By "system architecture" we mean the structure of the system (list of instances and their topology and interconnections). This assumes that so-called reflexive architectures are used, *i.e.,* architectures carrying a structural model of themselves. This is for example the case in our context, where this trask is referred to as *network discovery.*

Having (a), (b), and (c) allows to construct automatically the system model $(\mathcal{A}_i)_{i\in I}$ and even generate and deploy the monitoring algorithm automatically [1].

6.2 Probabilistic true concurrency models

In real-life applications, monitoring and diagnosis generally yield ambiguous results. For example, in real-life systems, multiple faults must be considered; as a result, it is often possible to explain the same observations by either one single fault or two independent faults. This motivates considering probabilistic models and developing maximum likelihood algorithms.

In doing this, we would obviously like that noninteracting subsystems are probabilistically independent. None of the classical probabilistic DES models (Markov chains, Hidden Markov Models, Stochastic Petri nets, stochastic automata) has this property. Samy Abbes [2, 3] has developed the fundamentals of true concurrency probabilistic models.

6.3 Timed true concurrency models

In performing monitoring or diagnosis, physical time (even imprecise) can be used to filter out some configurations. *Timed systems* models are needed for this. Candidates are timed automata and concurrent or partial order versions thereof [9].

6.4 Dynamically changing systems—Problem 6

So far we mentioned this problem but did not address it in this paper. In fact, addressing it is the very motivation for considering run-based on-line algorithms in which no diagnoser is statically pre-computed. Models of dynamically changing DES are not classical. A variety of them have been proposed in the context of distributed systems. *Petri net systems* [13] are systems of equations relating Petri nets; these models allow for dynamic instantiation of predefined nets. Variants of such models exist in the Petri net litterature. *Graph Grammars* [27] are more powerful as they use a uniform framework to represent both the movement of tokens in a net and the creation/deletion of transitions or subnets in a dynamic net. Graph Grammars have been used by Haar et al. [22] for diagnosis under dynamic reconfiguration. This subject is still in its infancy.

6.5 Incomplete models

For large, real-life systems, having an exact model (*i.e.,* accepting all observed runs while being at the same time non trivial) can hardly be expected. The kind of algorithm the DES community develops gets stuck when no explanation is found for an observation. In contrast, pattern matching techniques such as *chronicle recognition* [12] developed in the AI community are less precise than the DES model based techniques but do not suffer from this drawback. Leveraging the advantages of DES model based techniques to accepting incomplete models is a challenge that must be addressed.

7 Conclusion

We have discussed diagnosis of large networked systems. Our research agenda and requirements setting were motivated by the context of our ongoing cooperation with Alcatel, as briefly reported in the appendix. The focus of this paper was on on-line distributed diagnosis, where diagnosis is reported in the form of a set of hidden state histories explaining the recorded alarm sequences. In this context, efficiency of data structures to represent sets of histories is a key issue.

We have tried to deviate least possible from the classical setting, where distributed systems are modeled through the parallel composition of automata or languages. Our conclusion is that, to a certain extend, adopting a partial order viewpoint cannot be avoided. To the least, distributed executions must be seen as a partial order of interacting concurrent sequences of events. Of course, adopting a truly concurrent setting in which executions are systematically represented as partial orders is also possible.

This heterodox viewpoint raises a number of nonstandard research issues, some of which were listed in the previous section. While our group has started addressing some of these, much room remains for further research in this exciting area.

Another important remark we like to state is the usefulness of categorical techniques in analysing the issues we discussed in this paper. Note that we have considered a large variety of data structures to represent sets of runs. For each of them, we have considered the wished set of basic operators. Getting the desired factorization properties can become a real nightmare if only pedestrian techniques are used—see, *e.g.,* [17] for such a situation. In contrast, taking a categorical perspective [23] significantly helps structuring the research problems and focusing on the right properties for checking. It also prevents the researcher from redoing variants of her proofs. See for instance [4, 18, 19].

A Appendix, application context: distributed fault management in telecommunications networks

The techniques reported in this paper were developed in the context of a cooperation with the group of Armen Aghasaryan at Alcatel Research and Innovation. A demonstrator has been developed for distributed fault diagnosis and alarm correlation within the ALMAP Alcatel MAnagement Platform.

More recently, an exploratory development has been performed by Armen Aghasaryan and Eric Fabre for the Optical Systems business division of Alcatel. The system considered is shown in Fig. A.1. In this application, diagnosis is still performed centrally, but the system for monitoring is clearly widely distributed. Diagnosis covers both the transmission system (optical fiber, optical components) and the computer equipment itself. Fault propagation was not very complex but self-modeling proved essential in this context. Performance of the algorithms was essential.

A typical use case of distributed monitoring is illustrated if Figs. A.2–4. Fig. A.2 illustrates cross-domain management and impact analysis. The network for monitoring is the optical ring of Paris area with its four supervision centers. When a fault is diagnosed, its possible impact on the services deployed over it is computed—this is another kind of model based algorithm.

As for the optical ring itself, Fig. A.3 shows the system for monitoring. It is a network of several hundreds of small automata—called *managed objects*—having a handful of states and interacting asynchronously. Due to the object oriented nature of this software system, each managed object possesses its own monitoring system. This monitoring system detects failures to deliver proper service; it receives, from neighboring components, messages indicating failure to deliver service and sends failure messages to neighbors in case of incorrect functioning. This object oriented monitoring system causes a large number of redundant alarms travelling within the management system and subsequently recorded by the supervisor(s). Fig. A.4 shows a typical fault propagation scenario involving both horizontal (across physical devices) and vertical (across management layer hierarchy) propagation.

The problem of recognising causally related alarms is called *alarm correlation.* Fig. A.5 shows how monitoring results are returned to the operator, by proposing candidate correlations between the thousands of alarms recorded, *i.e.,* which alarm causally results from which other alarm. This shows by the way that diagnosis is not necessarily formulated, in real life applications, as that of isolating specific pre-defined faults.

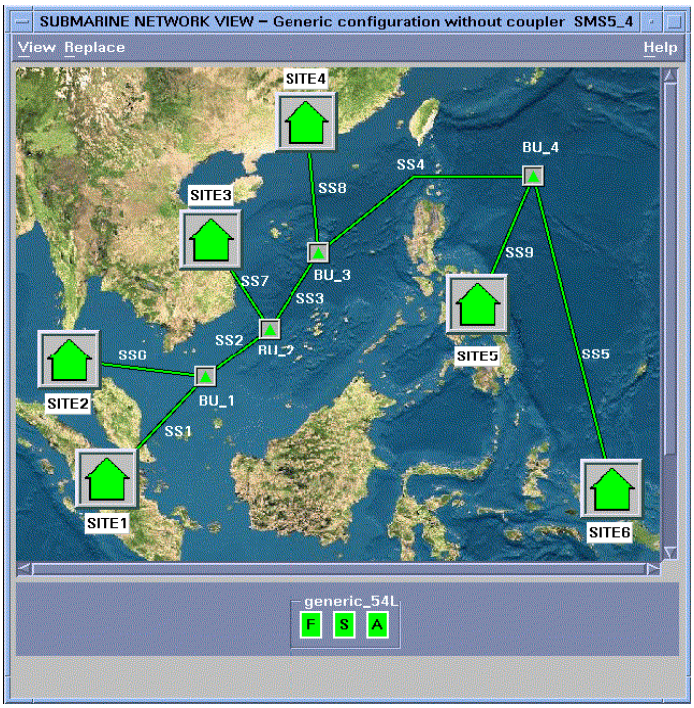

Figure A.1: the submarine optical telecommunication system considered for the trial with Alcatel Optical Systems business division and Alcatel Research and Innovation.

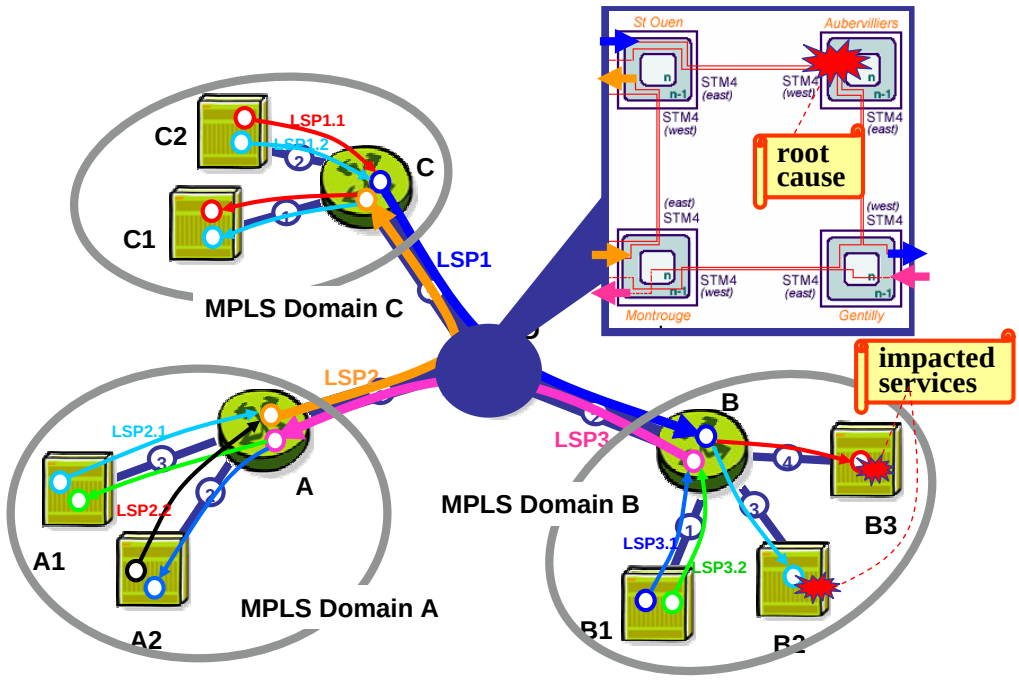

Figure A.2: failure impact analysis.

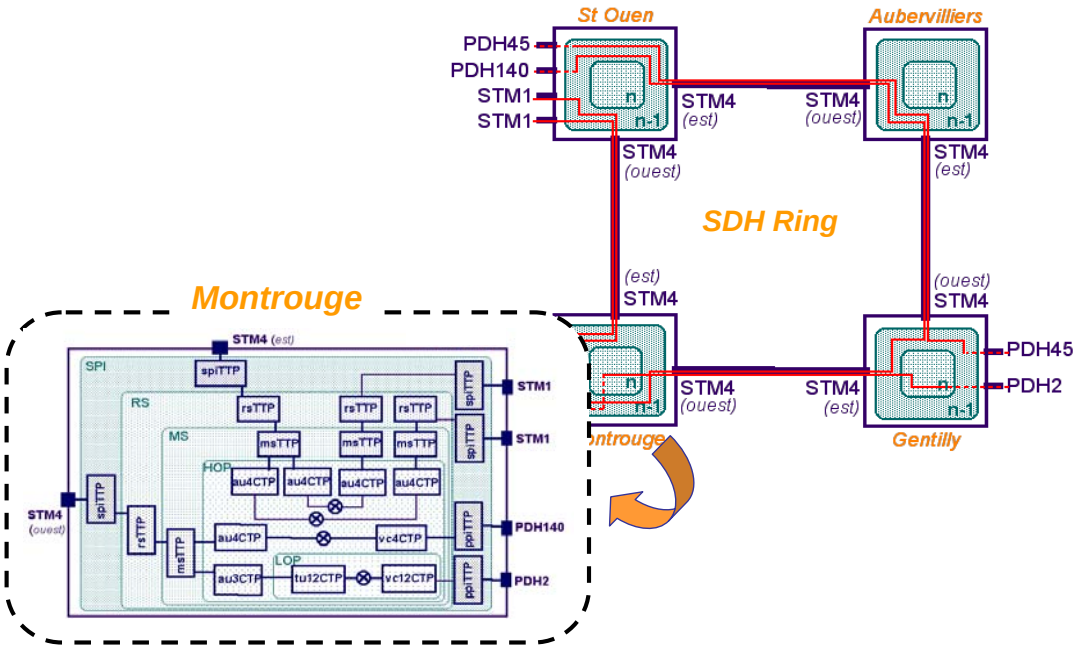

Figure A.3: the SDH/SONET optical ring of the Paris area, with its four nodes. The diagram on the left zooms on the structure of the management software, and shows its Managed Objects

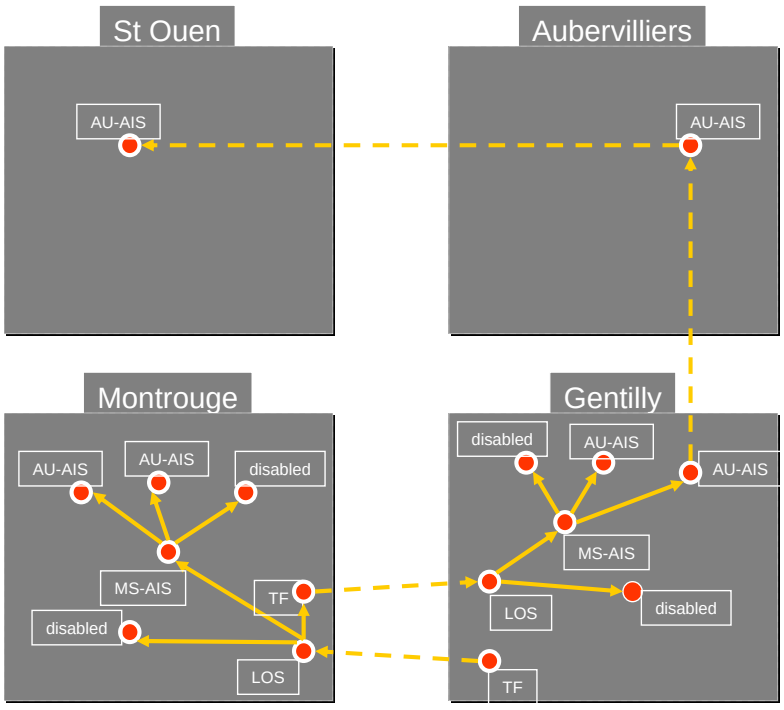

Figure A.4: showing a failure propagation scenario, across management layers (vertically) and network nodes (horizontally).

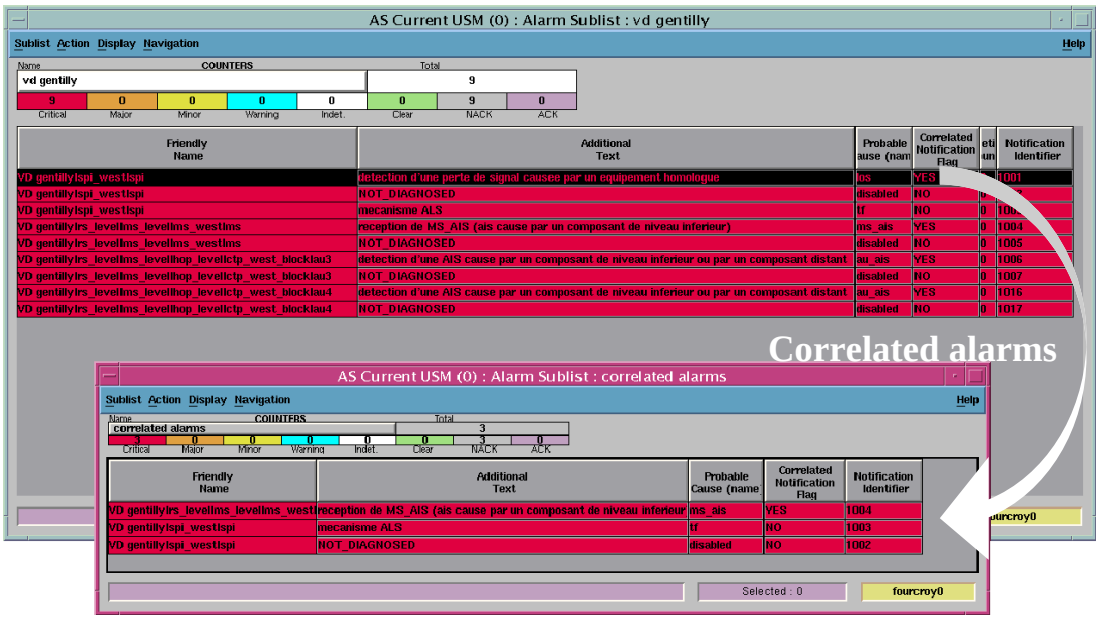

Figure A.5: returning alarm correlation information to the operator.

References

[1] A. Aghasaryan, C. Jard, J. Thomas. UML Specification of a Generic Model for Fault Diagnosis of Telecommunication Networks. In International Communication Conference (ICT), LNCS 3124, Pages 841-847, Fortaleza, Brasil, August 2004.

[2] S. Abbes, A. Benveniste. Branching Cells as Local States for Event Structures and Nets: Probabilistic Applications, in: FoSSaCSV. Sassone (editor), 2005, vol. 3441, pp. 95–109.

[3] S. Abbes and A. Benveniste. True-concurrency Probabilistic Models: Branching cells and Distributed Probabilities for Event Structures. *Information and Computation*, 204 (2), 231-274. Feb 2006.

[4] P. Baldan, S. Haar, and B. König. Distributed Unfolding of Petri Nets. Proc. of FOSSACS 2006, LNCS 3921, pp 126-141, Springer 2006.

[5] P. Baroni, G. Lamperti, P. Pogliano, M. Zanella, Diagnosis of Large Active Systems, Artificial Intell. 110, pp. 135-183, 1999.

[6] A. Benveniste, E. Fabre, S. Haar, C. Jard, Diagnosis of asynchronous discrete event systems, a net unfolding approach, *IEEE Trans. on Automatic Control,* vol. 48, no. 5, pp. 714-727, May 2003.

[7] R.K. Boel, J.H. van Schuppen, Decentralized Failure Diagnosis for Discrete Event Systems with Costly Communication between Diagnosers, in Proc. 6th Int. Workshop on Discrete Event Systems, WODES'02, pp. 175-181, 2002.

[8] R.K. Boel, G. Jiroveanu, Distributed Contextual Diagnosis for very Large Systems, in Proc. of WODES'04, pp. 343-348, 2004.

[9] T. Chatain, C. Jard. Time Supervision of Concurrent Systems using Symbolic Unfoldings of Time Petri Nets, in: 3rd International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS 2005), Springer Verlag, September 2005, LNCS 3829, p. 196–210.

[10] O. Contant, S. Lafortune, Diagnosis of Modular Discrete Event Systems, in Proc. of WODES'04, pp. 337-342, 2004

[11] R. Debouk, S. Lafortune, D. Teneketzis, Coordinated Decentralized Protocols for Failure Diagnosis of Discrete Event Systems, *J. Discrete Event Dynamic Systems,* vol. 10(1/2), pp. 33-86, 2000.

[12] Christophe Dousson, Paul Gaborit, Malik Ghallab: Situation Recognition: Representation and Algorithms. IJCAI 1993: 166-174

[13] R. Devillers and H. Klaudel. Solving Petri Net Recursions Through Finite Representation. Proc of IASTED'04.

[14] E. Fabre, Factorization of Unfoldings for Distributed Tile Systems, Part 1 : Limited Interaction Case, Inria research report no. 4829, April 2003. http://www.inria.fr/rrrt/rr-4829.html

[15] E. Fabre, Factorization of Unfoldings for Distributed Tile Systems, Part 2 : General Case, Inria research report no. 5186, May 2004. http://www.inria.fr/rrrt/rr-5186.html

[16] E. Fabre, Convergence of the turbo algorithm for systems defined by local constraints, Irisa research report no. PI 1510, 2003. http://www.irisa.fr/doccenter/publis/PI/2003/irisapublication.2006-01-27.8249793876

[17] E. Fabre, A. Benveniste, S. Haar, C. Jard, Distributed Monitoring of Concurrent and Asynchronous Systems, *J. Discrete Event Dynamic Systems,* special issue, vol. 15 no. 1, pp. 33-84, March 2005.

[18] E. Fabre, Distributed diagnosis based on trellis processes, in Proc. Conf. on Decision and Control, Sevilla, Dec. 2005, pp. 6329-6334.

[19] E. Fabre, C. Hadjicostis. A trellis notion for distributed system diagnosis with sequential semantics. In Proc. of Wodes 2006, Ann Arbor, USA, July 10-12, 2006.

[20] C.J. Fidge. Logical time in distributed computing systems. *IEEE Computer* **24**(8), 28–33, 1991.

[21] S. Genc, S. Lafortune, Distributed Diagnosis Of Discrete-Event Systems Using Petri Nets, in proc. 24th Int. Conf. on Applications and Theory of Petri Nets, LNCS 2679, pp. 316-336, June, 2003.

[22] S. Haar, A. Benveniste, E. Fabre, C. Jard. Fault Diagnosis for Distributed Asynchronous Dynamically Reconfigured Discrete Event Systems, in: IFAC World Congress Praha 2005, 2005.

[23] Saunders Mac Lane. *Categories for the Working Mathematician.* Springer Verlag, 1998.

[24] F. Mattern. Virtual time and global states of distributed systems, Proc. Int. Workshop on Parallel and Distributed Algorithms Bonas, France, Oct. 1988, Cosnard, Quinton, Raynal, and Robert Eds., North Holland, 1989.

[25] Y. Pencole, M-O. Cordier, L. Roze, A decentralized model-based diagnostic tool for complex systems. Int. J. on Artif. Intel. Tools, World Scientific Publishing Comp., vol. 11(3), pp. 327-346, 2002.

[26] W. Qiu and R. Kumar. A New Protocol for Distributed Diagnosis, 2006 American Control Conference, Minneapolis, June 2006.

[27] G. Rozenberg (ed.) *Handbook on Graph Grammars and Computing by Graph Transformation 1 (Foundations),* World Scientific, 1997.

[28] R. Su, Distributed Diagnosis for Discrete-Event Systems, PhD Thesis, Dept. of Elec. and Comp. Eng., Univ. of Toronto, June 2004.

[29] R. Su, W.M. Wonham, J. Kurien, X. Koutsoukos, Distributed Diagnosis for Qualitative Systems, in Proc. 6th Int. Workshop on Discrete Event Systems, WODES'02, pp. 169-174, 2002.

[30] R. Su, W.M. Wonham, Hierarchical Fault Diagnosis for Discrete-Event Systems under Global Consistency, *J. Discrete Event Dynamic Systems,* vol. 16(1), pp. 39-70, Jan. 2006.

[31] S. Tripakis. Undecidable Problems of Decentralized Observation and Control. In IEEE Conference on Decision and Control, 2001.

[32] T. Yoo, S. Lafortune, A General Architecture for Decentralized Supervisory Control of Discrete-Event Systems, *J. Discrete Event Dynamic Systems*, vol. 12(3), pp. 335-377, July, 2002.